The Mentat Programming Language and Architecture †

Andrew S. Grimshaw and Jane W. S. Liu
1304 W. Springfield Avenue
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

This paper describes Mentat, an object-oriented macro data-flow system. The objective of Mentat is to provide an easy-to-use, transparent mechanism to exploit parallelism. Mentat meets these objectives by combining a data-driven computation model, the macro data-flow model, with the object-oriented programming paradigm. In this paper we provide a high-level view of Mentat including the macro data-flow model, Mentat's graph representation mechanism called future lists, the Mentat programming language, the Mentat virtual machine, and the status of the implementation of a prototype Mentat system.

1. Introduction

In recent years, there has been an increased interest in exploiting new architectures that offer higher performance via parallelism. The new architectures include distributed systems such as those built around wide-band local networks, multicomputer systems such as hypercubes [1], and multiprocessor systems such as the Encore Multimax [2] and the Balance Sequent [3]. While the hardware capabilities of these systems have increased significantly, the software methodologies needed to program them have not advanced as rapidly.

This paper describes an object-oriented, macro data-flow system, called **Mentat**. The objective of Mentat is to provide an easy-to-use, transparent mechanism to exploit parallelism. Toward this end we formulated three major design goals. The Mentat system would: 1) support high degrees of parallelism, 2) not require a complex or centralized control mechanism, and 3) make it easy to program distributed and parallel applications. The primary innovations of Mentat are threefold. First, we have combined a data-driven computation model with the object-oriented programming paradigm. Second, we have developed a method for representing program graphs that supports dynamic graphs and permits decentralized control. Third, we have developed a mechanism transparent to the programmer that automatically detects data flow at run-time and constructs dynamic program graphs for programs written in a sequential object-oriented language.

In this paper we present a high level view of the Mentat system and discuss preliminary results and progress made to date. In Section 2 the principle features of the Mentat design are discussed: the macro data-flow model of computation, future lists, and the Mentat programming language. An example is presented to illustrate how data flow is automatically detected at run-time. Section 3 discusses an abstract view of the system architecture. Section 4 discusses the Mentat virtual machine that has been constructed and preliminary results obtained by executing Gaussian elimination on the machine. Section 5 contains a summary and discusses the future direction of research.

2. Mentat Overview

In this section we briefly present the macro data-flow model, the model of computation upon which Mentat is based. We then present futures and future lists, the graph representation technique used by Mentat, and describe how program objects are

† This work was partially supported by NASA Contract No. NAG 1-613.

mapped onto macro data-flow actors. The programming language used to construct Mentat programs is an extended C++ [4]. The aspects of the Mentat language that distinguish it from C++ are discussed.

2.1. Macro Data Flow

To satisfy the design goals of Mentat, we needed a model of computation that would support an appropriate degree of parallelism and not require a complex control mechanism. The data-flow model [5–10] was initially examined because of the high degrees of parallelism that can be easily achieved and because of the intuitive ease with which it maps onto message passing systems. However, the traditional data-flow model's shortcomings, small computation granularity [11,12] and determinism, quickly became apparent. These two shortcomings led us to propose the macro data-flow model as an alternative to the traditional data-flow model.

The macro data-flow model [13–15] is an extended data-flow model with three principle differences. First, the granularity of the actors is considerably larger. This provides the flexibility to choose an appropriate degree of parallelism. Second, some actors can maintain state information between firings. These actors are called *persistent actors.* Persistent actors provide an effective way to model side effects and non-determinism. Third, the structure of macro data-flow program graphs is not fixed at compile time. Instead, graphs can grow by the run-time elaboration of graph nodes into arbitrary subgraphs.

Macro actors are large-grain actors that perform high-level functions such as Gaussian elimination or FFT instead of individual machine instructions. The important characteristic of macro actors is that they are sufficiently computationally intensive, i.e., large grain. How computationally intensive is sufficient depends on the speed and latency of the communication channels. If the communication channels are relatively slow, then very large grains of computation are required. If the communication channels are very fast, smaller grains may suffice.

Some of the macro actors are *regular actors.* They are the same as actors in the traditional data-flow model. Specifically, all regular actors of a given type are functionally equivalent. A regular actor is enabled and may execute when all its input tokens are available. It performs some computation, generating output tokens that depend only on its input tokens. It may maintain internal state information during the course of a single execution. Information is saved during the performance of a function in much the same manner that scratch registers are used during a hardware multiply. However, no state information is preserved from one execution to another; they are pure functions.

Some macro actors are *persistent actors.* A persistent actor maintains state information that is preserved from one execution to the next. Hence the output tokens generated by a persistent actor during different executions are not necessarily the same for the same input tokens. Since different instances of a particular persistent actor type can be in different internal states, they are not identical.

A macro data-flow graph is a high-level view of a program. Nodes in this graph are macro actors. Hereafter, by actors, we mean macro actors. There is an arc from one actor to another when there is data dependency between them. However, by providing persistent actors, arcs are no longer the only way to specify data dependencies between computation primitives carried out by the system. Persistent actors provide us with a way to model information transfer between actors in different programs. The concept of persistent actors is similar to that of monitors [16] and resource managers [17].

2.2. Program Graphs and Future Lists

In all systems based on the data-flow model of computation some mechanism is needed to represent program graphs and to control their execution. Traditional data-flow systems are either static or dynamic [5–10].

In static systems the topology of the data-flow graph, and hence the execution pattern, is fixed at compile time. Dynamic systems allow multiple copies of a particular subgraph to be instantiated at run-time, e.g., multiple loop iterations or recursion. Thus, in a dynamic system multiple instances of the same subgraph may be executing concurrently. The dynamic instantiation of subgraphs is often implemented by labeling tokens with iteration or instance information. However, in both static and dynamic systems, the overall topology of each data-flow graph is fixed and is generated at compile time. This approach is adequate for traditional data-flow programs because each actor forwards its output tokens to a predetermined set of actors.

When designing the program graph representation and control scheme for Mentat we looked for a mechanism that would support: 1) decentralized control, 2) context-independence, and 3) dynamic binding and elaboration of vertices into nested subgraphs. Decentralized control is important in order to avoid the bottleneck a centralized control point would create. However, an overly complex distributed control can also place a heavy burden on the interconnection network, as well as the local hosts. We wanted a fairly simple decentralized scheme in which program graphs need not be fully replicated, and program subgraphs could be scheduled for remote execution without requiring any further control by the scheduling agent. For example, host **A** should be able to schedule subgraph **G** on host **B** and then be unconcerned as to how or where host **B** chooses to execute the subgraph **G**. The graph representation and control scheme should also allow actors to execute in a context-independent manner: actors need not be aware of from whom their arguments come, or to where their results are to be sent. This contrasts with systems and languages such as Occam [18] or CSP [19] in which entities are fully aware of their communication partners. Context-independence lets actors be more readily reusable.

Program graphs with static topologies are inadequate for macro data-flow systems and object-oriented languages for three reasons. The first reason is that macro data-flow actors may be persistent and unbound at compile time. Each instance of a persistent actor must be differentiated from all other instances, and it may not be possible to know until run time which instance of a persistent actor is used by the program. Therefore, it is not possible to associate each node of the graph with a particular instance of the required actor. Enumerating all possible instances is not feasible because the instance actually used may not exist at compile or load time.

The second reason that static topologies are inadequate follows from our use of the object-oriented paradigm. Static topology implies that the types of objects can be determined at compile time so that the subgraphs that implement their function can be included in the program graph. However, in object-oriented languages, it will often not be possible to know even the type of object in use. For instance, in Smalltalk [20], we know only that the object supports some method. Exactly how the method is implemented, i.e., its subgraph, cannot be determined until run-time when its type is known. Thus we must be able to delay the binding of subgraphs for method implementation until run-time. Providing dynamic graph elaboration of actors into subgraphs satisfies this need.

The third reason is that when the topology of the graph is static the granularity of computation is reduced, and the performance will suffer. To see why the granularity is reduced we examine how control is performed in static graphs. Certain actors in the program graph, called *control actors*, determine at run-time the actual execution pattern for the graph, e.g., which subgraphs are executed, and how many loop iterations or recursive calls are invoked. Examples of control actors include greater than, less than, equal to, switches, and merges. If static graphs

are used we must use control actors to determine the actual execution pattern. Control actors have very small granularity, often with only a single instruction. There can be many control actors even for simple operations such as unrolling a loop or performing conditional and case statements. Further, we must pay the communication and scheduling overhead for each control actor executed just as we do for larger grain actors. We can eliminate much of the control overhead by moving the control of subgraph selection and unrolling into actors and by allowing the actors to construct subgraphs at run-time that reflect the effects of performing the control function internally.

To represent program graphs that satisfy our goals of decentralized control, context independence, and dynamic elaboration with delayed binding, Mentat introduces a new implementation technique for macro data-flow called *futures* and *future* lists. Each actor receives a list of futures with its input tokens. The list of futures describes what happens after the actor has completed its computation. It is a set of directed data-flow graphs for which the actor is the greatest common ancestor. Each future in the list of futures describes a directed graph that corresponds to a particular outgoing arc from the actor. A *future list* is a list of futures. Mentat futures should not be confused with Multilisp futures [21]. Futures are similar in concept to continuations [22, 23]. Unlike a Scheme continuation, which specifies an ordered sequence of events, a future describes a directed graph with many possible parallel paths.

Each future has two components, an *arc_instance* and a *future list*. Each arc_instance describes an arc between two nodes in a subgraph. Since the source is implicit, the arc_instance is the name of a destination node that is to receive a copy of the output token. In particular the arc_instance contains information about which instance of the destination actor is to be used and to which input of that actor the arc is connected. Because the same subgraph may be used many times, possibly concurrently, the arc_instance also contains a *computation tag* that specifies which instantiation of the graph the token is for. Computation tags are similar to token colorings [5]. In addition to the arc_instance, each future also contains a *future list*.

```
future_list  ==  future |
                 future future_list |
                 passive | ϵ
future       ==  (arc_instance future_list)
```

Upon completion of the the actor computation, an output token containing the result and the appropriate future list is sent to each arc_instance named in each future of the actor's future list. When an actor requires two or more tokens, all but one of the received future lists are redundant. The use of a passive future list in a message indicates that some other token to the actor carries the future list for the actor. Passive future lists are usually short.

Figure 1 illustrates the correspondence between a future list and a portion of a program graph. **A** receives the following future list:

(<B, op1,arg1> (<D, op1,arg1> ϵ))
(<C, op1,arg1> (<D, op1,arg2> ϵ)) ϵ

A sends a copy of its results to <B,op1,arg1> and <C,op1,arg1>. **B** receives a future list of (<D,op1,arg1> ϵ), and **C** receives a future list of (<D,op1,arg2> ϵ). **B** and **C**, upon completion, send their results to **D**. Beyond **D** the future is not

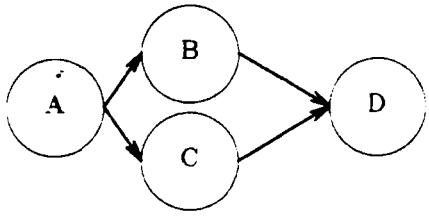

Figure 1. Correspondence to Graphs

specified.

An actor may construct new future lists, augment its own future list, and start new subgraphs with the constructed future lists. The new subgraph that results from these operations is called the *elaborated subgraph* of the actor. The elaboration into subgraphs is completely hidden from users of the actor. The elaborated subgraph of an operation is not necessarily the same every time the operation is executed.

Elaboration of the program graph can be accomplished in one of two ways. First, an actor can construct a future list that consists of the specification of the elaborated subgraph and the current future list. Its results are forwarded to the elaborated subgraph, and the results of the elaborated subgraph are forwarded to its future. The second method involves the construction of a new elaborated subgraph in which the output of the new subgraph is sent to the current future list. The new elaborated subgraph may include initial tokens on the arcs. The initial tokens are provided by the constructing node.

The use of future lists as a graph description mechanism allows graph control to be completely decentralized. Each actor receives enough of the program graph to continue the computation. There is no need to coordinate the execution of separate subgraphs, and their execution may proceed independently. Furthermore, it is not necessary for the entire program graph to be generated at compile time. Indeed, the structure of the graph is only implied, and changes as actors modify their future lists.

Future lists can also be used to provide alternative paths of execution in the event that special conditions arise. For example, suppose that there is a database server actor and the actor is passed several future lists. The first is used after the successful completion of the database operation. The second is used for recovery if the database operation fails. The third is used if a security violation is detected. Each of the future lists represents a subgraph to be invoked by the database server on behalf of the caller when the corresponding condition is true at completion. Invocation of a subgraph is equivalent to sending tokens to the source nodes of the subgraph and sending the future lists that define the subgraph with the tokens.

2.3. Mapping Objects to Macro Data Flow

An object-oriented approach to software design and implementation provides the ideal support for the design and development of macro data-flow programs. In Mentat, macro actors are realized as external operations of Mentat objects. This section discusses the relationship between objects and actors, the naming of objects and actors, and the construction of future lists.

2.3.1. Objects, Actors, and Tokens

Before describing Mentat objects, we must differentiate between independent and contained objects [24]. *Independent objects* are objects that have disjoint address spaces. *Contained objects* do not have disjoint address spaces. An independent object may contain many contained objects. Contained objects may communicate with one another and with the independent object containing them using shared memory via pointers or the stack. Independent objects may communicate with one another only via messages. Thus, if message passing is cheap, all objects should be independent objects. If message passing is expensive, only computationally complex objects should be independent. Mentat permits both independent objects and contained objects. However, objects declared as Mentat objects are independent objects.

Each instance of a Mentat object consists of four components: a name, a representation of the data stored in the object, a set of externally visible operations, and an independent thread of control. The set of externally visible operations is the object's interface. The actual implementation of an operation frequently makes use of other objects and operations. How the operations

are performed, either serially or in parallel, what other objects are used, and the internal representation of data structures are not visible to any outside object.

In Mentat, macro data-flow actors are an abstraction implemented by the operations of objects. For example, an encryption object would have a single operation that requires two parameters, text and key. It implements a single actor with two input arcs, one for each parameter. A queue object may have five operations: full, empty, clear, enqueue, and dequeue, each corresponding to an actor. These operations could easily be implemented without referencing other Mentat objects. In general a Mentat object operation may be implemented using other object operations. In this case an elaborated subgraph is constructed.

An operation of a Mentat object is invoked by sending a set of messages (tokens) to the object, one for each parameter. When all of the parameters for a particular operation have arrived the corresponding actor is enabled and the operation is executed. The marshaling of parameters and sending them as messages is transparent to the programmer.

2.4. The Mentat Language

One problem facing parallel systems designers is how to simplify the writing of parallel programs. Proposals range from automatic program transformation system such as Paraphrase [25] which extract parallelism from sequential programs, to the use of side-effect free languages [7,8], to the use of languages and systems where the programmer must explicitly manage all aspects of communication, synchronization, and parallelism [26,27]. The problem with fully automatic schemes is that they are best suited for detecting small grain parallelism. The problem with schemes in which the programmer is completely responsible for managing the parallel environment is that complexity can overwhelm the programmer.

Mentat provides a compromise solution. Programmers are responsible for identifying those objects that are of sufficient computational complexity to allow efficient parallel execution, and the compiler and run-time system are responsible for managing parallelism, communication, and synchronization. In order to exploit these capabilities programs must be written in the Mentat programming language. Rather than create a new programming language, the Mentat programming language is an extended C++ [4]. A preprocessor takes source code in the extended language and automatically generates code to perform run-time data-flow detection and macro data-flow graph construction. Users specify those object classes that are to be transformed into macro data-flow actors. Instances of these classes are used normally. The system will automatically generate the macro data-flow graphs for the program, reducing the programming effort required.

There are five principle extensions to the C++ language: Mentat classes, the member functions *create()* and *destroy()*, the return-to-future (*rtf()*) mechanism, the *select/accept* (guarded) statement, and the member function *main()* for each Mentat class. Below we examine each briefly. A more complete description of the language can be found in [15].

2.4.1. Mentat Class Definition

Mentat classes are the mechanism for specifying Mentat actors. Each actor is implemented by an operation of a Mentat object, and each Mentat object is composed of one or more actors. In C--, objects are defined by their class. Each class has an interface section in which member variables and member functions are defined. Not all class objects should be Mentat objects. In particular, objects that do not have a sufficiently high communication ratio, i.e., the object operations are not computationally complex enough, should not be Mentat objects. To provide the programmer a way to

control the degree of parallelism, Mentat allows both standard C++ classes and Mentat classes to be defined. By default, a standard C++ class definition defines a standard C++ object. Standard C++ objects and variables are contained objects contained in their lexically enclosing Mentat class. The programmer defines a Mentat class by using the keyword *MENTAT* in the class definition. He may specify whether the class is PERSISTENT or REGULAR, as in the following example.

```
PERSISTENT MENTAT class big_matrix {
   // private data and member functions
public:
   vector gauss_elim();
   ... more member functions
};
```

Persistent and regular class definitions correspond to persistent and regular objects.

2.4.2. Create and Destroy

To instantiate and destroy instances of Mentat objects we have added two new reserved member functions for all Mentat class objects: *create()* and *destroy()*. These functions are inherited from the base class Mentat and can be overloaded by the programmer of the class. The *create()* function is used to instantiate new instances of Mentat classes. It takes as parameters user-provided initialization information. *Create()* also allows the user the option of specifying where the new instance is to be instantiated, e.g., on a different processor, or on the same processor as another Mentat object. Thus, the programmer can give the underlying system information that will be useful in making instantiation decisions and thus influence where the new object is actually instantiated.

2.4.3. Return to Future (*rtf()*)

The function *rtf()* is the Mentat analog to the *return()* of C++. Its purpose is to allow Mentat member functions (actors) to return a value to the successor nodes in the macro data-flow graph in which the member function appears. The *rtf()* does not, however, mark the end of the actor computation.

Rtf() takes two types of arguments, local variables or constants, and subgraphs. Local variables must satisfy the same restrictions that apply to Mentat arguments: it must be possible to determine their length, and they must be contiguous in memory. Subgraphs are automatically generated, and their use is transparent to the programmer. Returning a subgraph using *rtf()* is the mechanism for actor subgraph elaboration.

The following example illustrates both how *rtf()* is used and how program graphs are automatically constructed. The code fragment for object operation *op1* appears in Figure 2. If *expression(input)* is TRUE, then *op1* will generate a token containing the value 5 that will be forwarded to *op1's* successor. In this case there is no subgraph elaboration. If, on the other hand, *expression(input)* is FALSE, then a subgraph will be generated for *op1* at run-time. The actual subgraph generated will depend on the value of *expression(local-variable)*. If it is TRUE, then the subgraph shown in Figure 3 will be generated for *op1* at run-time. The result of the new subgraph will be directed to the successor of *op1*. We would like to emphasize that the generation of subgraphs is entirely transparent to the programmer. The mechanism used to automatically generate subgraphs is discussed in more detail in [15].

```
if expression(input) { rtf(5); }
else {
   w=A.operation1(4,5);
   x=B.operation1(w,2);
   if expression(local-variable)
      y=C.operation1(4,w);
   else
      y=C.operation1(w,x);
   z=D.operation1(y,w,x);
   rtf(z);
};
```

Figure 2. Code for *op1*.

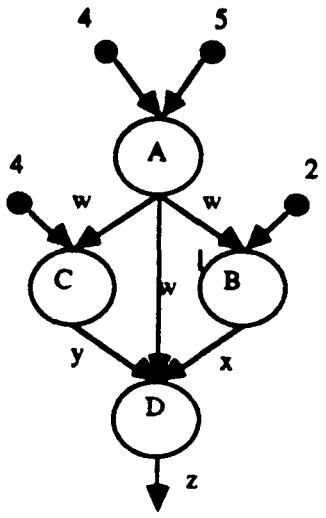

Figure 3. Subgraph generated at run-time.

2.4.4. Guards

The Mentat programming language has a select/accept statement that is similar to the ADA [28] select/accept. Unlike ADA guards, Mentat guards may contain the formal parameters of the member function being guarded and message tag information such as the sender or computation tag. Assignment statements are disallowed in guards to prevent side effects. This feature provides the ability to selectively receive messages based upon their contents and was inspired by PLITS [29].

Guards are evaluated in the order of their priority. Within a given priority level each of the guards is evaluated in some non-deterministic order. Each guard is evaluated in turn until one of the guards is true; the corresponding statement-list for that guard is then executed. When the statement-list associated with the guard has been executed, control passes to the next statement beyond the select.

2.4.5. The Member Function *Main()*

The member function *main()* is a reserved function name for Mentat classes. It is the initial thread of control for new instances of a Mentat class. The function body may be any sequence of extended C++ statements. Usually *main()* will consist of an initial select/accept to accept the create call for the class, if one exists. Then, once the create has been executed, a loop is entered in which there is a select statement with an accept for each member function of the class. If no *main()* is provided by the programmer one will be generated for the class.

3. The Mentat Virtual Machine

The Mentat virtual machine is an idealized machine for executing macro data-flow programs on a variety of hardware architectures. The virtual machine presents the image of a single logical machine to programmers. The virtual machine is a three-level abstraction. The highest level is very similar to traditional data-flow machines. The two lower levels of the abstraction permit modeling of single CPU systems, shared memory multiprocessers, loosely coupled systems (e.g., hypercube), and combinations of the three as specific instances of the model. Various components of the machine are responsible for tasks that support token matching, object instantiation, object management, scheduling, communication, and actor computation.

3.1. The System Level

The highest level of abstraction is the system level. The system level of abstraction defines an abstract machine that is similar to traditional data-flow hardware architectures. There are three components of the abstraction: the interconnection network, the computation units (called), and the matching unit & token store, as shown in Figure 4. The three components perform the same functions as in traditional architectures. In traditional architectures the matching unit matches tokens for actors and sends the actors with their tokens via the interconnection network to the computation units. The computation units execute the actor, and return the results to the matching unit via the interconnection network. The corresponding Mentat virtual machine components are described next.

The *global matching unit* is a single logical entity with two principle functions:

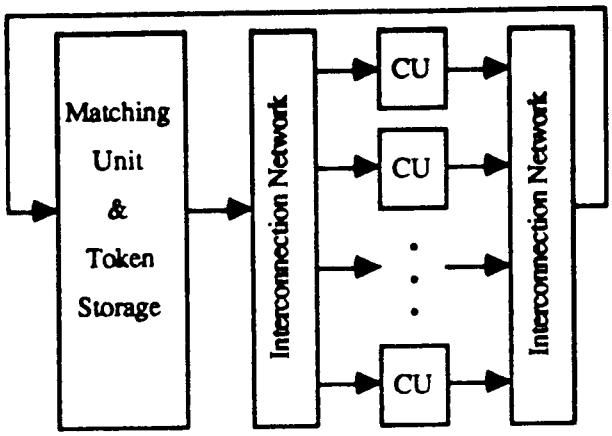

Figure 4. System Level of Abstraction.

first, to store state objects, code objects, and tokens; second, to match tokens and construct *work units* for the computation units. The global matching unit matches tokens and determines when an actor is ready to fire. Enabled actors, with their tokens and code, are packaged into work units. A work unit (WU) is a tuple (code, state, actor number, token-list). A work unit contains all of the information needed by a computation unit to execute an actor: code, state, and input tokens. The work units are sent via the interconnection network to an idle computation unit. The global matching unit receives an initial set of tokens and state objects to begin the matching task. Then, as computation proceeds, tokens and state objects are consumed and new ones are received from the computation units.

The *interconnection network* provides communication services between the computation units and the global matching unit. At this level of abstraction all computation units are equidistant from the global matching unit. The communication is error free with guaranteed delivery.

The *computation units* (CU) accept work units from the global matching unit, perform the computation, and send the results back to the global matching unit. They are pure functional units. The output of a computation unit depends solely on the contents of the work unit. Thus any computation unit may be used to execute any work unit. When a computation is complete the computation unit constructs a *result package* and sends it via the interconnection network to the global matching unit. The result package is a tuple (state, token-list). The state is the new state for the actor that was just executed. The token-list is a list of output tokens that the actor generated. Once the result package has been sent the computation unit is ready to accept another work unit.

The problem with the system level of abstraction just presented is the centralized nature of the global matching unit. All token matching, storage, and actor scheduling is done by a single entity. If the global matching unit (or its communication channels) becomes overloaded then there will be idle computation units and the performance of the system will suffer. In a distributed system such a bottle neck is undesirable. Since Mentat is designed for distributed systems, this level of abstraction is not appropriate. The global matching unit should appear to be a single entity, but it can actually be a distributed entity. Further, the machine should take advantage of localities. It is much cheaper, for instance, to send a work unit to some computation units that others. These issues provide the motivation for the middle level of abstraction.

3.2. The Processor Level

The idealized macro data-flow machine is as shown in Figure 4. However, in distributed systems, the matching unit is fragmented, and the cost of communicating from a particular portion of the matching unit to different computation units varies. The fragmentation of the matching unit is accounted for in the processor level of abstraction shown in Figure 5. This middle level more accurately reflects actual hardware configurations likely to be found in a distributed system.

In the processor level there are many *local matching units* (LMUs). Each local

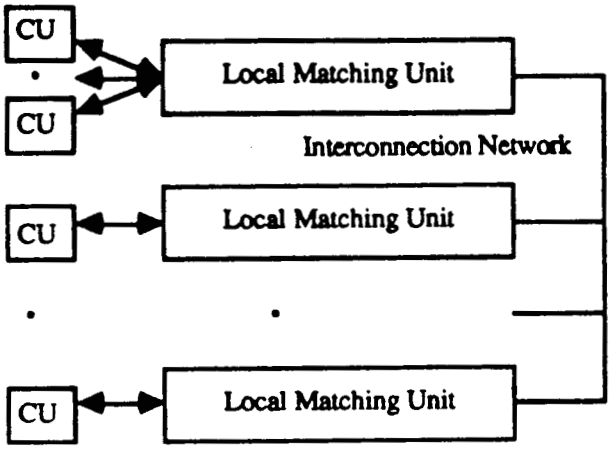

Figure 5. Intermediate Level of Abstraction.

matching unit has associated with it one or more computation units. The existence of more than one associated computation unit could indicate either that multiple virtual computation units are mapped onto a single processor, or that one of the nodes is a shared memory multiprocessor such as the Encore Multimax. By definition, it is cheaper in time and/or resource use for a local matching unit to communicate with its associated computation unit(s) than with other computation units.

The function of the global matching unit in the system level is actually performed in a cooperative manner by the local matching units in the processor level. Thus, in addition to sending work units to, and receiving result packages from, called, the individual local matching units must communicate and coordinate with each other. Hence, local matching units are more complex than the global matching unit.

3.3. The Local Machine Level

The lowest level of the abstraction consists of the actual implementation of the LMU's. The LMU itself consists of many components: the local message handler, the inter-LMU message handler, and the token matchers for instantiated and uninstantiated objects that implement the global matching unit. The token matcher for un-instantiated objects in particular is complicated by the distributed nature of the matching problem: how can we know where a matching token resides, or if it has even been generated yet? For a more complete description of the LMU implementation see [15].

4. Status

Mentat has been implemented on a ten-processor Encore Multimax. The implementation consists of two parts, the programming support and a virtual machine. Implemented Mentat tools include a preprocessor that transforms extended C++ programs to C++ source or object code and a set of library routines accessible to the preprocessor that provides an interface to the virtual machine.

The virtual machine executes macro data-flow programs that have been prepared using the preprocessor. The prototype can simulate a wide variety of Mentat configurations. The number of hosts, the number of processors on each host, the interconnection topology between hosts, and the speed of the communication links can all be specified. Thus, one can simulate Mentat executing on a hypercube, a mesh, or a single shared bus system.

To demonstrate the performance of Mentat we executed Gaussian elimination using the partial pivoting method as a benchmark. Two versions were prepared. The first is a serial version written in C and executed in a single Unix process. The second was written for Mentat execution using persistent actors. Both programs were run on matrices of dimension 100, 200, 300, 400, and 500. Figure 6 shows the execution times for the serial version, and for Mentat configured as a 4 LMU hypercube, a 6 LMU shared bus, and an 8 LMU hypercube. The Mentat versions are clearly superior, particularly in the larger dimensions. The corresponding speed-ups are shown in Figure 7.

Graph Labels

-+- Serial
... 2-D Hypercube
- - - 6 processor bus
—— 3-D Hypercube

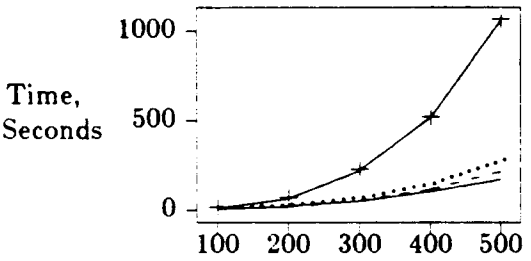

Figure 6

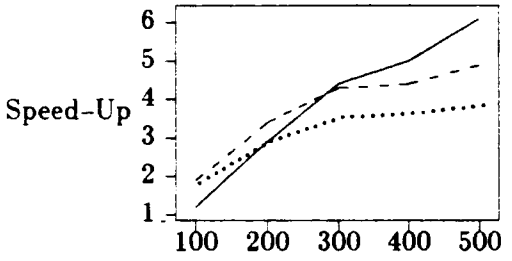

Figure 7

The primary cause of the low speed-ups observed in the lower dimensions is the relatively high communication ratio, particularly in the eight LMU hypercube version where the tokens have to travel up to three hops. The communication ratio is also affected by the machine overhead and run-time graph construction overhead. Both grow linearly with the dimension of the matrix while the amount of computation is $O(n^3)$. Thus, as the dimension of the matrix increases the communication ratio decreases.

5. Summary

In this paper we presented a high level overview of the object-oriented macro data-flow system called Mentat. We discussed the design goals of Mentat, the macro data-flow model of computation, future lists, the Mentat programming language, and the Mentat virtual machine.

The preliminary results on performance of Mentat are encouraging. The next step is to develop a full scale Mentat simulator that provides accurate timing predictions for large numbers of virtual processors. Then, if simulation results show continued improvement, we will implement Mentat on an actual distributed system of SUN's, IBM's, or Macintosh II's. Such a distributed version would provide the most convincing evidence of the efficacy of the Mentat approach.

Concurrent with the development of an improved simulator an actual application should be developed using the Mentat language. This is important for two reasons. First, since ease of programming is a design goal, we must evaluate how difficult Mentat is to program for non-specialists, i.e., people not involved in the Mentat design. Second, we must demonstrate that Mentat can exploit significant parallelism in real-world applications. Before any system can move out of the lab it must be shown to be useful on non-contrived problems. Thus we must implement an actual application such as ray-tracing or a Monte-Carlo simulation. The implementation of actual applications would realistically demonstrate the effectiveness of the Mentat approach.

References

[1] Charles L., "The Cosmic Cube," *Communications of the ACM*, pp.22-33,vol. 28, 1985.

[2] *UMAX 4.3 Release Notes*, Encore Computer Corporation, Marlbrough, Massachusetts, 1987.

[3] Osterhaug, Anita,"GUIDE TO PARALLEL PROGRAMMING On Sequent Computer Systems,"*Sequent Technical Publications*, Sequent Computer Systems, Beaverton, OR, 1986.

[4] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[5] Dennis, J., "First Version of a Data Flow Procedure Language," MIT TR-673, May, 1975.

[6] Agerwala, T., and Arvind, "Data Flow Systems," *IEEE Computer*, vol. 15, no. 2, pp. 10–13, February, 1982.

[7] Ackerman, W. B., "Data Flow Languages," *IEEE Computer*, vol. 15, no. 2, pp. 15–25, February, 1982.

[8] McGraw, James R.,"The VAL Language: Description and Analysis," *ACM Transactions on Programming Languages and Systems*, pp. 44–82, vol. 4, no. 1, January, 1982.

[9] Srini, V. P., "An Architectural Comparison of Dataflow Systems," *IEEE Computer*, pp. 68–88, March, 1986.

[10] Veen, Arthur H., "Dataflow Machine Architecture," *ACM Computing Surveys*, pp. 365–396, vol. 18, no. 4, December, 1986.

[11] Babb, R. F., "Parallel Processing with Large-Grain Data Flow Techniques," *IEEE Computer*, pp. 55–61, July, 1984.

[12] Gaudiot, J. L., and M. D. Ercegovac, "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," *Proceedings of the 1984 IEEE Conference on Distributed Systems*, 1984.

[13] Liu, J. W. S., and A. S. Grimshaw, "A Distributed System Architecture Based on Macro Data Flow Model," *Proceedings Workshop on Future Directions in Architecture and Software*, South Carolina, May 7–9, 1986.

[14] Liu, J. W. S., and A. S. Grimshaw, "An object–oriented macro data flow architecture," *Proceedings of the 1986 National Communications Forum*, September, 1986.

[15] Grimshaw, Andrew S., "Mentat: An Object–Oriented Macro Data Flow System," University of Illinois, TR UIUCDCS–R–88–1440, June, 1988.

[16] Hoare, C.A.R.,"Monitors: An Operating System Structuring Concept," *Communications of the ACM* pp.549–557, vol. 17, no. 10, October, 1974

[17] Arvind and J. D. Brock, "Resource Managers in Functional Programming," *Journal of Parallel and Distributed Computing*, vol.1, pp. 5–21, 1984.

[18] *Occam Programming Manual*, Inmos Ltd, Prentice–Hall, New York, 1984.

[19] Hoare, C.A.R.,"Communicating Sequential Processes," *Communications of the ACM*, pp. 666–677, August, 1978.

[20] Goldberg, A., and D. Robson, *Smalltalk–80: The Language and its Implementation*, Addison–Wesley, Reading, MA, 1983.

[21] Halstead, Robert H. Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, pp. 501–538, vol. 7, no. 4, October, 1985.

[22] Abelson, H., Sussman, G. J., and J. Sussman, "Structure and Interpretation of Computer Programs," The MIT Press, Cambridge Massachusetts, 1985.

[23] Stoy, J. E., "Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory," The MIT Press, Cambridge, Massachusetts, 1977.

[24] Nierstrasz, O.M.,"Hybrid: A Unified Object–Oriented System," *IEEE Database Engineering*, vol. 8, no. 4, pp. 49–57, December, 1985.

[25] Kuck, D., Kuhn, R., Leasure, B., Padua, D., and M Wolfe, "Dependence Graphs and Compiler Optimizations," *ACM Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pp. 207–218, January, 1981.

[26] Andrews, Gregory R., and Fred B. Schneider, "Concepts and Notions for Concurrent Programming," *ACM Computing Surveys*, pp. 3–44, vol. 15, no. 1, March, 1983.

[27] Filman, Robert E., and Daniel P. Friedman, *COORDINATED COMPUTING Tools and Techniques for Distributed Software*, McGraw–Hill Book Company, New York, 1984.

[28] *Reference Manual for the Ada Programming Language*, United States Department of Defense, Ada Joint Program Office, July 1982.

[29] Feldman, J. A.,"High Level Programming for Distributed Computing," *Communications of the ACM*, pp. 353–368, vol. 22, no. 6, January, 1979.